Dokumentacja końcowa TKOM

Marcel Jarosz 304006

```
Wybrany wariant ma:
- statyczne typowanie
- silne typowanie
- zmienne są domyślnie stałe
- zmienne mogą być opcjonalne
```

Opis projektu

Projekt zakłada wykonanie interpretera własnego języka ogólnego przeznaczenia, który będzie zawierał implementację `pattern matching`

Możliwości języka

Język jest typowany silnie i statycznie. Domyślnie wszystkie zmienne są tworzone jako `const`. Istnieje możliwość nadania opcjonalności danej zmiennej, przez co może ona mieć wartość `null`.

- Obsługa kilku standardowych typów danych (bool, int, double, string)
- Zmienne mogą być zdefiniowane jako opcjonalne (operator `?` przy nazwie typu)
- Istnieje możliwość utworzenia zmiennej jako `mutable`
- Istnieje możliwość tworzenia jednolinijkowych komentarzy (`#comment`)
- Typ znakowy `string` obsługuje escapowanie i konkatenację
- Obsługa operatorów matematycznych i logicznych według priorytetów, nawiasowanie
- Możliwość używania instrukcji warunkowych (`if [else]`) oraz pętli (`while`)
- Możliwość definiowania własnych funkcji oraz ich wywoływanie (również rekursywne)
- Ograniczony stos wywołań funkcji
- Argumenty do funkcji są przekazywane przez wartość
- Tworzone zmienne mają swoje określone zakresy (np. tylko wewnątrz funkcji)
- Zmienne utworzone w kontekście globalnym są widoczne z każdego miejsca
- Możliwość użycia funkcji dopasowania wzorca dzieki konstrukcji `match(wyrażenie)`
- Operator `??`, który umożliwia wykonanie pewnego działania zależnie czy wyrażenie posiada wartość czy nie
- Obsługa operatora `as` służącego do rzutowania wyrażeń na inne typy wbudowane
- Obsługa operatora `is` pozwalającego sprawdzić typ danego wyrażenia i/lub posiadanie przez niego wartości

Operatory

Możliwa jest budowa złożonych wyrażeń używających operatorów z predefiniowanymi priorytetami. Działania na operatorach o takich samych priorytetach będą wykonywane od lewej do prawej. Operatory według priorytetów:

1. Negacja - `!, -`
2. Operatory mnożenia

- iloczyn - `a * b`

- iloraz - `a / b`
- iloraz całkowity - `a // b`
- reszta z dzielenia - `a % b`

3. Operatory addytywne

- dodawanie - `a + b`
- odejmowanie - `a - b`

4. Operatory typów

- sprawdzenie typu/obecności wartości - `a is int`
- rzutowanie - `a as double`

5. Operatory relacyjne

- porządkowe - `a < b, a <= b, a > b, a >= b`
- równości - `a == b, a != b`

6. Operator logiczny koniunkcji - `a and b`
7. Operator logiczny alternatywy - `a or b`

Słowa kluczowe

as, and, bool, break, continue, double, else, false, func, if, int, is, match, mutable, null, or, return, string, true, void, while

Gramatyka

```
program               = progStatement, { progStatement } ;

progStatement         = functionDef
                      | statement ;

statement             = conditionalStatement
                      | simpleStatement, ";"
                      | variableDeclaration, ";" ;

conditionalStatement  = ifStatement
                      | whileStatement
                      | matchStatement;

simpleStatement       = expression
                      | returnStatement
                      | "break"
                      | "continue" ;

functionDef           = "func", identifier, "(", [parametersList], ")", ":",
type, "{", statementBlock, "}" ;

statementBlock        = statement, {statement} ;
variableDeclaration   = ["mutable"], type, identifier, assignmentOp, (expression)
;

ifStatement           = ifBlock, [elseBlock] ;
```

```
whileStatement        = "while", "(", expression, ")", "{", {statementBlock}, "}"
;
matchStatement        = "match", "(", expression, ")", "{", insideMatchStatement,
{insideMatchStatement}. "}" ;

insideMatchStatement  = (insideMatchExpression, {(orOp | andOp),
insideMatchExpression} | "default"), "=>" simpleStatement, "," ;

insideMatchExpression = comparisonOp, valueLiteral
                      | expression
                      | "is", (type | "null");

ifBlock               = "if", "(", expression, ")", "{", {statementBlock}, "}" ;
elseBlock             = "else", "{", {statementBlock}, "}" ;

functionCall          = identifier, "(", [argumentList], ")" ;
parametersList        = parameter, {",", parameter} ;

argumentList          = expression, {",", expression} ;

parameter             = ["mutable"], type, identifier ;
returnStatement       = "return", [expression] ;

expression            = nullCheckExpression, ["=", nullCheckExpression] ;
nullCheckExpression   = orExpression, {"??", orExpression} ;
orExpression          = andExpression, {orOp, andExpression} ;
andExpression         = compExpression, {andOp, compExpression} ;
compExpression        = isasExpression, [comparisonOp, isasExpression] ;
isasExpression        = addExpression, [("is" | "as"), (type | "null")] ;
addExpression         = mulExpression, {additiveOp, mulExpression} ;
mulExpression         = unaryExpression, {multiOp, unaryExpression} ;
unaryExpression       = [unaryOp], baseExpression ;
baseExpression        = valueLiteral
                      | "(", expression, ")"
                      | functionCall
                      | identifier ;

(* operators *)

unaryOp               = "!"
                      | "-" ;
assignmentOp          = "=" ;
orOp                  = "or" ;
andOp                 = "and" ;
comparisonOp          = "=="
                      | "!="
                      | "<"
                      | "<="
                      | ">"
                      | ">=" ;
additiveOp            = "+"
                      | "-" ;
multiOp               = "*"
                      | "/"
                      | "//"
```

```
                      | "%" ;


valueLiteral          = booleanLiteral
                      | integerLiteral
                      | doubleLiteral
                      | stringLiteral
                      | "null" ;

nonNullableType       = "bool"
                      | "int"
                      | "double"
                      | "string" ;

type                  = nonNullableType, ["?"]
                      | "void" ;

booleanLiteral        = "true"
                      | "false" ;

identifier            = (letter | "_"),  letter, {digit | letter | "_"} ;

escapeLiteral         = "\\" ("t" | "b" | "r" | "n" | "\"" | "\\") ;
stringLiteral         = "\"", {charLiteral}, "\"" ;
charLiteral           = allCharacters - "\\" - "\"" | escapeLiteral ;

doubleLiteral         = integerLiteral, ".", { digit }
                      | ".", digit, {digit};

integerLiteral        = "0" | naturalLiteral ;
naturalLiteral        = digitNoZero, { digit } ;

allCharacters         = ? all visible characters ? | "\t" | " " ;
letter                = "a".."z" | "A".."Z" ;

digit                 = "0"
                      | digitNoZero ;
digitNoZero           = "1" | ... | "9" ;
```

Wymagania funkcjonalne interpretera

- Możliwość wczytania, parsowania i analizy fragmentów kodu zapisanych w plikach tekstowych
- Wykonywanie wszystkich poprawnie zapisanych instrukcji
- Informowanie o wystąpieniu błędów w odpowiednich etapach analizy
- Wyświetlanie wyników działania skryptu na wyjściu standardowym

Wymagania niefunkcjonalne interpretera

- Wyświetlane komunikaty o popełnionych błędach zawierają nr linii i kolumnę w którym zostały napotkane.

Środowisko uruchomieniowe i wybrane technologie

Projekt został utworzony w środowisku Java w wersji 17. Do zarządzania zależnościami został wybrany Maven, a testy zostały zrealizowane przy użyciu biblioteki JUnit.

Obsługa interpretera

Program jest aplikacją konsolową uruchamianą wraz z odpowiednimi parametrami (np. ścieżka do skryptu)
Wyniki działającego programu jak i napotkane błędy będą wypisywane na standardowym wyjściu.

```
java -jar tkom-interpreter.jar path_to_file
```

Budowa i działanie

Podział na moduły

- ładowanie źródła
- lekser
- parser
- analizator semantyczny
- executor

Analiza leksykalna

Analizator leksykalny ma za zadanie podzielić wczytane źródło na pojedyncze tokeny. Dokonuje tego poprzed odczyt zawartości znak po znaku, do czasu odczytania całości sekwencji odpowiadającej poprawnemu tokenowi języka.

Parser

Parser ma za zadanie sprawdzić czy dostarczane do niego tokeny są ułożone według zdefiniowanej wcześniej gramatyki. Kolejne tokeny będą decydować o wybranej ścieżce wgłąb analizowanego programu (parser rekursywnie zstępujący). Następnie zostanie utworzone drzewo składniowe (AST), które pozwoli na dalszą analizę skryptu.

Analizator semantyczny

Moduł ten jest odpowiedzialny za weryfikację poprawności znaczenia wygenerowanego drzewa składniowego.
Jest on odpowiedzialny m. in. za:

- sprawdzenie unikalności identyfikatorów w danym kontekście
- sprawdzenie unikalości identyfikatorów funkcji
- zgodność operacji arytmetycznych i logicznych
- weryfikacja typów

Podczas weryfikacji znaczenia semantycznego drzewa składniowego tworzy on nowe drzewo obiektów, które zostanie użyte potem do wykonania programu.

Moduł wykonawczy

Odpowiada za poprawne wykonanie wszystkich instrukcji zawartych w drzewie wytworzonym w poprzednich etapach. Po wykonaniu wszystkich instrukcji program kończy swoje działanie.

Funkcje wbudowane

- `print(string text) -> void` - odpowiedzialna za wypisywanie podanych argumentów na ekran
- `get_input() -> string` - umożliwia pobranie tekstu podanego przez użykownika na standardowym wejściu;

Testowanie

Do modułu analizatora leksykalnego zostały utworzone testy jednostkowe.
Dla modułu analizatora składniowego zostały utworzone testy integracyjne łączące działanie leksera i korzystającego z niego parsera.

Moduł analizy semantycznej oraz wykonania posiada przygotowane testy akceptacyjne na zadanych fragmentach kodu.

Zostały przygotowane testy akceptacyjne, do których wejściem będą fragmenty kodu napisane w omawianym języku.

Przykłady

Przykład 1

```
# function calculates Nth fibonacci number
func fib(int n) : int {
    if (n <= 1) {
        return n;
    }
    return fib(n - 2) + fib(n - 1);
}

mutable int i = 1;
int value = 13;
mutable double? sum = null;

while (i <= 10) {
    sum = (sum ?? 0.0) + (fib(i) * value as double);
    i = i + 1;
}

string resMessage = "Sum is: " + (sum as string);
print(resMessage);
```

Przykład 2

```
func even(int value): bool {
    return value % 2 == 0;
}

func odd_and_divisible(int value, int div): bool {
    return value % 2 == 1 and value % div == 0;
}

string userInput = get_input();

match(userInput as int?) {     # return value of this expression can be accessed
via '_'
    is int and even(_ as int) => print("Number" + (_ as string) + " is even"),
    is int and odd_and_divisible(_ as int, 3) => print("Number" +  (_ as string)
+ " is odd and divisible by 3"),
    default => print("Is not a number"),
}
```